**Data Structure Theory: Advanced Trees**

|  |
| --- |
| **Data Structure: AVL Tree** |
| **Definition:** An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property. |
| **Complexities:** |

|  |
| --- |
| **Data Structure: Red Black Tree** |
| **Definition:** A **red–black tree** is a [self-balancing binary search tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree). Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node. These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.  Tracking the color of each node requires only 1 bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to a classic (uncolored) [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree). |
| **Complexities:** |

|  |
| --- |
| **Data Structure: B - Tree** |
| **Definition:** A **B-tree** is a self-balancing N-Ary search [tree](https://en.wikipedia.org/wiki/Tree_data_structure) . The B-tree is a generalization of a [binary search tree](https://en.wikipedia.org/wiki/Binary_search_tree) but each node can have more than two children. B-trees are optimized for systems that read and write large blocks of data. B-trees are a good example of a data structure for external memory. It is commonly used in [databases](https://en.wikipedia.org/wiki/Database) and [filesystems](https://en.wikipedia.org/wiki/Filesystem).  To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc ) require O(h) disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc. |
| **Complexities:** |

**Red Black Tree Vs. AVL Tree (Deciding Which to Use):**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Insert** | **Deletion** | **Look Up Operation** |
| **Small Data Size** | **Red Black Tree** will be faster because on avarge it uses less tree rotations | **Red Black Tree** will be faster because on avarge it uses less tree rotations | **AVL Tree** is faster, because AVL tree has less depth and height. |
| **Large Data Size** | **AVL Tree** is faster. because you need to lookup for a particular node before insertion. As you have more data the time difference on looking up the particular node grows but AVL tree & RB tree still only need constant number of rotation at the worst case. Thus the bottle neck will become the time you lookup for that particular node. | **AVL Tree** is faster. because you need to lookup for a particular node before insertion. As you have more data the time difference on looking up the particular node grows but AVL tree & RB tree still only need constant number of rotation at the worst case. Thus the bottle neck will become the time you lookup for that particular node. | **AVL Tree** is faster, because AVL tree has less depth and height. |